Claudio De Sio Cesari

Il nuovo Java

Operatori e gestione del flusso di esecuzione (NJ–004)

https://www.nuovojava.it

Operatori e gestione del flusso di esecuzione

- Conoscere e saper utilizzare i vari operatori (unità 4.1).
- Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2).
- Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.3).
- Conoscere e saper utilizzare il nuovo costrutto switch (unità 4.4).
- Comprendere che prima di iniziare a programmare è necessario prima avere chiari tutti i requisiti, e avere un piano per procedere nella codifica (unità 4.5)

Operatori di base

- Operatori binari ed unari (simboli o lettere)
- Operatore d'assegnazione =

```
int variabile1 = 0;
int variabile1 = 1;
int variabile2 = 2;
variabile1 = variabile2;
```

- Operatori aritmetici: **+ – * / %** (**modulo**)

L'espressione 5 % 3 restituisce 2
L'espressione 10 % 2 restituisce 0
L'espressione 100 % 50 restituisce 0
L'espressione 5.5 % 3.3 restituisce 2.2

- Gli operatori + e – sono anche unari:

```
int i = -1;
int j = +1;
```

Operatori di assegnazione composti

Descrizione	Operatore
Somma ed assegnazione	+=
Sottrazione ed assegnazione	-=
Moltiplicazione ed assegnazione	*=
Divisione ed assegnazione	/=
Modulo ed assegnazione	%=

Per esempio se abbiamo:

```
int i = 5;
```

Le seguenti istruzioni sono equivalenti:

```
i = i + 2;
i += 2;
```

Infatti, questa verrà trasformata dal compilatore nell'istruzione:

```
i = (i + 2);
```

Operatori (unari) di pre e post-incremento (e decremento)

Descrizione	Operatore	Esempio
Pre-incremento di un'unità	++	++i
Pre-decremento di un'unità	--	--i
Post-incremento di un'unità	++	i++
Post-decremento di un'unità	--	i--

Le seguenti istruzioni sono tutte equivalenti

```
i = i + 1;
```

oppure:

```
i += 1;
```

ma anche:

```
i++;
```

oppure:

```
++i;
```

Pre-incremento:

```
int x = 5;        // x= 5
int y = ++x;      //x = 6, y = 6
```

Post-incremento:

```
x = 5;            // x= 5
y = x++;          // x= 6, y= 5
```

Operatori relazionali o di confronto

Operatore	Simbolo	Applicabilità
Uguale a	==	Tutti i tipi
Diverso da	!=	Tutti i tipi
Maggiore	>	Solo i tipi numerici
Minore	<	Solo i tipi numerici
Maggiore o uguale	>=	Solo i tipi numerici
Minore o uguale	<=	Solo i tipi numerici

- Non confondere l'operatore di assegnazione = con l'operatore ==

```
boolean b = (9 == 9); // b vale true
int i = 10;
double d = 10.0;
boolean b1 = (i != d); // b1 vale false
boolean b2 = b1 == (i >= 11); // b2 vale true
boolean b2 = false == (false)
```

Operatori relazionali e reference

Se consideriamo la seguente classe:

```
public class Alunno {
   public String nome;
   public Alunno(String n) {
      nome = n;
   }
}
```

il seguente snippet:

```
Alunno alunno1 = new Alunno("Simone");
Alunno alunno2 = new Alunno("Simone");
System.out.println(alunno1 == alunno2);
```

Stamperà *false*.

Operatori logico - booleani

Descrizione	Operatore
NOT logico	!
AND logico	&
OR logico	|
XOR logico	^
Short circuit AND	&&
Short circuit OR	||
AND e assegnazione	&=
OR e assegnazione	|=
XOR e assegnazione	^=

```
boolean b1 = !true;  // b1 vale false
boolean b2 = !b1;    // b2 vale true
b2 = !b2;            // b2 vale false
```

Operatori logico - booleani

Operando1	Operando2	Op1 AND Op2	Op1 OR Op2	Op1 XOR Op2
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false

```
boolean and = true & false;  // and è false
boolean or = and | !false;   // or è true
boolean xor = and ^ or;      // xor è true
and = or & xor;              // and è true
or = !(xor | and);           // or è false
xor = (and ^ !or);           // xor è false;
boolean flag = ( (a != 0) && (b/a > 10) )      // short circuit
boolean flag = ( (a == 0) || (b/a > 10) )      // short circuit
```

Concatenazione di stringhe con +

Il simbolo dell'operatore aritmetico di addizione + può essere usato anche come operatore per concatenare stringhe. Per esempio, il seguente frammento di codice:

```
String nome = "James";
String cognome = "Gosling";
String nomeCompleto = "Mr. " + nome + " " + cognome;
```

farà in modo che la stringa *nomeCompleto*, abbia come valore "Mr. James Gosling".

Se "sommiamo" un qualsiasi tipo di dato con una stringa, il tipo di dato sarà automaticamente convertito in stringa, e ciò può spesso risultare utile. Bisogna però stare attenti ad alcune situazioni. Per esempio:

```
System.out.println(1 + 1 + "3" + 7);
```

stamperà:

```
237
```

Priorità degli operatori

separatori	. [] () ; ,
da sx a dx	++ -- + - ~ ! (tipo_di_dato)
da sx a dx	* / %
da sx a dx	+ -
da sx a dx	<< >> >>>
da sx a dx	< > <= >= instanceof
da sx a dx	== !=
da sx a dx	&
da sx a dx	^
da sx a dx	|
da sx a dx	&&
da sx a dx	||
da dx a sx	?:
da dx a sx	= *= /= %= += -= <<= >>= >>>= &= ^= |=

Operatori bitwise

Java eredita in blocco tutti gli operatori del linguaggio C, un linguaggio nato per altri scopi in un'epoca diversa. Alcuni degli operatori quindi, sono completamente ignorati dalla maggior parte dei programmatori Java. In particolare, gli **operatori bitwise** che permettono di eseguire operazioni direttamente sui bit, sono quasi inutili in Java, visto che trattare la memoria a basso livello non è tra i compiti assegnati alla programmazione Java. Abbiamo quindi deciso di non spiegare questi operatori in queste pagine, spostandoli nell'apposito approfondimento 4.2 online.

Costrutti di programmazione semplici

- **Condizioni** (o strutture di controllo decisionali): permettono, durante la fase di runtime, una scelta tra l'esecuzione di istruzioni diverse, a seconda che sia verificata o meno una specificata condizione. Tale condizione, coincide con il risultato di un'operazione booleana.
- **Cicli** (o strutture di controllo iterative): consentono, in fase di runtime, di decidere il numero di esecuzioni di determinate istruzioni.
- Java, definisce essenzialmente due condizioni: il costrutto **if** ed il costrutto **switch**), cui si aggiunge l'operatore ternario. I costrutti di tipo ciclo invece sono quattro: **while**, **for**, **do** (detto anche **do-while**) e il **ciclo for migliorato** (detto anche **foreach**).

Il costrutto condizionale *if*

Il costrutto **if**, è una condizione che permette di prendere semplici decisioni basate su un'espressione booleana. Un'**espressione booleana** è un'espressione che restituisce solo valori di tipo *boolean*, vale a dire *true* o *false*. Essa di solito si avvale di operatori di confronto e, se necessario, di operatori logici. La sintassi del costrutto è la seguente:

```
if (espressione-booleana) {
   istruzione_1;
   istruzione_2;
   ……;
   istruzione_i;
}
```

Nel caso di un'unica istruzione, è possibile omettere le parentesi graffe:

```
if (espressione-booleana)
   istruzione;
```

Il costrutto condizionale *if*

Esempio:

```
if (numeroLati == 3)
   System.out.println("Questo è un triangolo");
```

è equivalente a:

```
if (numeroLati == 3) {
   System.out.println("Questo è un triangolo");
}
```

Nell'esempio l'istruzione di stampa verrebbe eseguita se e solo se la variabile *numeroLati* avesse valore *3*. In quel caso, l'espressione booleana *numeroLati == 3* varrebbe *true*, e quindi sarebbe eseguita l'istruzione di stampa.

Se invece l'espressione restituisse *false*, sarebbe eseguita direttamente la prima istruzione che segue il costrutto.

Il costrutto condizionale *if*

Possiamo anche estendere la potenzialità del costrutto *if* mediante la parola chiave *else*:

```
if (espressione-booleana){
   istruzione_1;
   istruzione_2;
   ……;
   istruzione_i;
} else {
   istruzione_i+1;
   istruzione_i+2;
   ……;
   istruzione_n;
}
```

Il costrutto condizionale *if*

Esempio:

```
if (numeroLati == 3) {
   System.out.println("Questo è un triangolo");
} else {
   System.out.println("Questo non è un triangolo");
}
```

La parola *if* si traduce in italiano con la parola *se*; la parola *else* invece con *altrimenti*, nel senso di *ogni altro caso*. Quindi, se l'espressione booleana è vera verrà stampata la stringa "Questo è un triangolo"; se è falsa verrà stampata la stringa "Questo non è un triangolo".

Se volessimo tradurre in italiano il precedente codice, potremmo scrivere:

```
se il numero di lati è uguale a 3
   stampa "Questo è un triangolo"
altrimenti
   stampa "Questo non è un triangolo"
```

Il costrutto condizionale *if*

Possiamo anche comporre più costrutti nel seguente modo con clausole *else if*:

```
if (espressione-booleana1) {
   istruzione_1;
   istruzione_2;
   ……;
   istruzione_i;
} else if (espressione-booleana2) {
   istruzione_1+1;
   ……;
   istruzione_j;
} else if (espressione-booleana3) {
   istruzione_j+1;
   ……;
   istruzione_k;
} else {
   istruzione_k+1;
   ……;
   istruzione_n;
}
```

Il costrutto *while*

- Il ciclo *while* permette di eseguire più volte uno statement (o un insieme di statement compresi in un blocco di codice) tante volte fino a quando una certa condizione booleana è verificata.
- Ogni ciclo di esecuzione viene detta iterazione.
- La sintassi è la seguente:

```
[inizializzazione;]
while (espressione-booleana) {
   istruzione_1;
   istruzione_2;
   ……;
   istruzione_i;
   [aggiornamento iterazione;]
}
```

Il costrutto *while*

- Come esempio, presentiamo una piccola applicazione che stampa i primi dieci numeri:

```
public class WhileDemo {
   public static void main(String args[]) {
      int i = 1;
      while (i <= 10) {
         System.out.println(i);
         i++;
      }
   }
}
```

Costrutti di programmazione avanzati

- È vero che con *if* e *while* possiamo risolvere praticamente tutti i problemi della programmazione, ma è anche vero che a volte sono scomodi da utilizzare rispetto ad altri costrutti. Alcuni di quelli che vedremo sono infatti utilizzatissimi.
- Per esempio il ciclo ***for***, è molto più utilizzato del ciclo **while**. Anche il **ciclo for migliorato** (noto anche come **foreach**), è molto utilizzato, perché la sintassi è poco verbosa. Altri costrutti lo sono meno, come le istruzioni **break** e **continue**, l'operatore ternario, e il ciclo **do-while**, ma sono comunque fondamentali per programmare.
- Un discorso a parte sarà fatto per il costrutto condizionale **switch**, che ora può essere usato nella maniera tradizionale come statement ma anche come espressione. A partire dalla versione 12 di Java infatti, il costrutto è stato modernizzato, ed ora risulta molto più utile.

Il costrutto *for*

- Il ciclo ***for***, è probabilmente il ciclo più completo che offre il linguaggio. Di seguito trovate la sua sintassi nel caso d'utilizzo di una o più istruzioni da iterare.
- Uno statement:

```
for (inizializzazione; espressione booleana; aggiornamento)
   statement;
```

- Più statement:

```
for (inizializzazione; espressione booleana; aggiornamento) {
   statement_1;
   ……;
   statement_i;
}
```

Il costrutto *for*

```
public class ForDemo {
   public static void main(String args[]) {
      for (int n = 10; n > 0; n--) {
         System.out.println(n);
      }
   }
}
```

- Sintassi compatta
- Variabile locale al ciclo

Il costrutto *for*

```
public void forMethod(int j) {
   int i;
   for (i = 0; i < j; ++i) {
      System.out.println(i);
   }
   System.out.println("Numero iterazioni = " + i);
}
```

- Da Java 10 possiamo usare anche *var*:

```
String [] strings  = {"Antonio", "Ludwig",
   "Johann Sebastian", "Piotr"};
for (var i = 0; i < strings.length; i++) {
   System.out.println(strings[i]);
}
```

for o *while*?

- Il costrutto *for* è più utilizzato quando ci sono indici:

```
char alfabeto[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l', m', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'z'} ;
for (int i = 0; i < 21; i++) {
   System.out.println(alfabeto[i]);
}
```

- Il while per gestire lo **stato di un oggetto** (**state basic logic**):

```
public class Interruttore {
   public static final int ACCESO = 0;
   public static final int SPENTO = 1;
   public int posizione;
   public void accendiLampadina() {
      while (interruttore.posizione == ACCESO) {
         // codice omesso
      }
   }
}
```

Il costrutto *do*

```
[inizializzazione;]
do {
   istruzione_1;
   istruzione_2;
   ……;
   istruzione_i;
   [aggiornamento iterazione;]
} while (espressione booleana);
```

- Ciclo meno utilizzato
- Almeno una iterazione viene sempre eseguita

Il costrutto *do*

L'output del seguente mini-programma:

```
public class DoWhile {
   public static void main(String args[]) {
      int i = 10;
      do {
         System.out.println(i);
      } while(i < 10);
   }
}
```

è:

```
10
```

Quindi la prima iterazione è stata comunque eseguita.

Ciclo *for* migliorato (foreach)

- Enhanced *for* (sarebbe meglio dire *semplificato*)

```
for (variabile_temporanea : oggetto_iterabile) {
   istruzione_1;
   istruzione_2;
   ……;
   istruzione_i;
}
```

Per esempio

```
int [] arr = {1,2,3,4,5,6,7,8,9};
for (int tmp : arr) {
   System.out.println(tmp);
}
```

Funziona sugli oggetti «iterable»

L'operatore ternario (o condizionale)

È un operatore che può sostituire a volte una condizione:

variabile = (espressione-booleana) ? espr1 : espr2;

che il tipo della variabile e quello restituito da *espr1* ed *espr2* siano compatibili.

Viene usato come espressione:

```
String query = "select * from table" +
   (condition != null ? " where " + condition : "");
```

La parola chiave *break*

La parola *break* può far terminare un qualsiasi ciclo:

```
int i = 0;
while (true) { //ciclo infinito
   if (i > 10) {
      break;
   }
   System.out.println(i);
   i++;
}
```

La parola chiave *continue*

La parola chiave *continue* fa terminare non l'intero ciclo, ma solo l'iterazione corrente (ovvero fa saltare alla prossima iterazione).

```
int i = 0;
do {
   i++;
   if (i == 5) {
      continue;
   }
   System.out.println(i);
} while(i <= 10);
```

Etichette (label)

```
int j = 1;
pippo: //possiamo dare un qualsiasi nome ad
        //una label
while (true) {
   while (true) {
      if (j > 10)
         break pippo;
      System.out.println(j);
      j++;
   }
}
```

Il nuovo *switch*

- Il costrutto *switch*, come tutti i costrutti di Java e l'operatore ternario, è stato ereditato dal linguaggio C sin dalla prima versione di Java (fa eccezione il ciclo *for* migliorato che è stato introdotto con Java 5)
- Il costrutto *switch* ha delle caratteristiche che ben si adattavano alla creazione dei tipici programmi che venivano sviluppati una volta in C, per esempio i parser e codificatori binari.
- Con la versione 12 è stato aggiornato usando feature preview
- Nuova sintassi, con una nuova parola chiave (*yield*) e un nuovo operatore (->)

Il nuovo *switch*

- Il costrutto *switch*, come tutti i costrutti di Java e l'operatore ternario, è stato ereditato dal linguaggio C sin dalla prima versione di Java (fa eccezione il ciclo *for* migliorato che è stato introdotto con Java 5)
- Ha delle caratteristiche che ben si adattavano alla creazione dei tipici programmi che venivano sviluppati una volta in C, per esempio i parser e codificatori binari.
- Con la versione 12 è stato aggiornato usando feature preview (approfondimento 4.4)
- Nuova sintassi, con una nuova parola chiave (*yield*) e un nuovo operatore (->)

Il nuovo *switch*

- È uno statement condizionale come lo è il costrutto *if*. Permette di eseguire determinate istruzioni piuttosto che altre, in base al valore che verrà passato al runtime al costrutto. Per esempio:

```
public void switchTest(byte test) {
   switch (test) {
      case 1:
         System.out.println("case 1");
      break;
      default:
         System.out.println("default");
      break;
   }
}
```

```
switch (test) {
   case valore_1:
      [istruzioni;]
   [break;]
   case valore_2:
      [istruzioni;]
   [break;]
   . . .
   case valore_n:
      [istruzioni;]
   [break;]
   [default:]
      [istruzioni;]
   [break;]
}
```

switch tradizionale: sintassi

test può essere di tipo:

- *byte*, *short*, *char*, *int*,
- *Byte*, *Short*, *Character*, *Integer*
- *String*
- Enumerazione

switch tradizionale: Fall through

```
public class SeasonSwitch {
   public static void main(String args[]) {
      Integer month = 4;
      String season;
      switch (month) {
         case 12:
         case 1:
         case 2:
            season = "winter";
            break;
         case 3:
         case 4:
         case 5:
            season = "spring";
            break;
//senza questo break: season = summer
         case 6:
         case 7:
         case 8:
            season = "summer";
            break;
         case 9:
         case 10:
         case 11:
            season = "autumn";
            break;
         default:
            season = "not identifiable";
            break;
      }
      System.out.println("The season is
       " + season);
   }
}
```

switch : Fall through con *enum*

```
import java.time.Month;
public class SeasonSwitchEnumTest {
   public static void main(String args[]) {
      Month month = Month.APRIL;
      String season;
       switch (month) {
         case DECEMBER:
         case JANUARY:
         case FEBRUARY:
            season = "winter";
            break;
         case MARCH:
         case APRIL:
         case MAY:
            season = "spring";
            break;
         case JUNE:
         case JULY:
         case AUGUST:
            season = "summer";
            break;
         case SEPTEMBER:
         case OCTOBER:
         case NOVEMBER:
            season = "autumn";
            break;
         default:
            season = "not identifiable";
            break;
      }
      System.out.println("The season is
       " + season);
   }
}
```

switch : Fall through con stringa

```
public String getTipoGiornoSettimana(String
    giornoDellaSettimana) {
  String typeOfDay;
  switch (giornoDellaSettimana) {
    case "Monday":
      typeOfDay =
        "Inizio settimana";
      break;
    case "Tuesday":
    case "Wednesday":
    case "Thursday":
      typeOfDay =
        "Settimana piena";
      break;
    case "Friday":
      typeOfDay = "Fine settimana
lavorativa";
      break;
    case "Saturday":
    case "Sunday":
      typeOfDay = "Weekend";
      break;
    default:
      typeOfDay = "Indefinito!";
      break;
  }
  return typeOfDay;
}
```

Il nuovo *switch*: notazione freccia

```
import java.time.Month;
public class SeasonSwitchStatementArrowEnumTest {
   public static void main(String args[]) {
      Month month = Month.APRIL;
      String season = null;
      switch (month) {
         case DECEMBER, JANUARY, FEBRUARY -> season = "winter";
         case MARCH, APRIL, MAY -> season = "spring";
         case JUNE, JULY, AUGUST -> season = "summer";
         case SEPTEMBER, OCTOBER, NOVEMBER -> season = "autumn";
      }
      System.out.println("The season is " + season);
   }
}
```

Il nuovo *switch*: notazione freccia

- I vari *case* possono dichiarare più label separate da virgole (il fall through non è necessario, anzi non può usare)
- Non c'è stato bisogno di utilizzare la parola chiave *break*
- La sintassi basata sulla notazione freccia, rende il nostro codice meno verboso, più chiaro, ed eliminando la possibilità di eseguire il fall through, anche più facile da gestire.
- Notare che per specificare più istruzioni dopo la notazione freccia, bisogna includerle in una coppia di parentesi graffe. per esempio:

```
switch (month) {
    case DECEMBER, JANUARY, FEBRUARY -> {
        season = "winter";
        System.out.println("OK");
    }
    // resto del codice omesso
```

Il nuovo *switch*: notazione freccia ed *enum*

```
public enum Colore {
   VERDE, GIALLO, ROSSO;
}
public class Semaforo {
   public void cambiaColore(Colore colore) {
      switch(colore) {
         case VERDE  ->  System.out.println("La luce è verde");
         case GIALLO ->  System.out.println("La luce è gialla");
         case ROSSO  ->  System.out.println("La luce è rossa");
      }
   }
}
```

Come usare il costrutto *switch*

- Usare il costrutto switch nella maniera più semplice e completa possibile (meglio utilizzare tutte le quattro parole chiave quando possibile), infatti:
  - Evitare il fall through (per la sua scarsa leggibilità e la difficoltà di gestione)
  - Per ogni case usare sempre il relativo *break*. Ancora meglio è utilizzare la notazione freccia ottenere lo stesso risultato in maniera più chiara e semplice
  - È sempre consigliabile usare una clausola *default*
  - È sempre consigliabile mantenere un ordine logico dei vari *case* per non incorrere in dimenticanze e peggiorare la leggibilità.
  - Anche per la clausola *default*, è consigliato utilizzare il *break*, pure se posizionata come ultima clausola di un costrutto *switch*.
  - Ricordiamo che le variabili locali, condividono il ciclo di vita con il blocco di codice in cui sono definite. Se definiamo una variabile locale all'interno di un *case* di uno *switch*, questa sarà visibile all'interno di tutti i *case* dello stesso costrutto dichiarati successivamente. Se si vuole evitare questo comportamento, è possibile creare blocchi di codice in maniera arbitraria all'interno di un *case*.

Espressione *switch*

- Con il termine **espressione**, intendiamo un'istruzione (un literal, un'invocazione di metodo, un'operazione etc.) che ritorna un valore. Quindi un'espressione *switch* è un costrutto che ritorna un valore. Trasformiamo l'esempio del fall through con un'espressione *switch*:

```
import java.time.Month;
public class SeasonSwitchExpressionEnumTest {
   public static void main(String args[]) {
      Month month = Month.APRIL;
      String season = switch(month) {
         case DECEMBER, JANUARY, FEBRUARY -> "winter";
         case MARCH, APRIL, MAY -> "spring";
         case JUNE, JULY, AUGUST -> "summer";
         case SEPTEMBER, OCTOBER, NOVEMBER -> "autumn";
      };
      System.out.println("The season is " + season);
```

Espressione *switch*

- Possiamo usare la nuova parola definita dal linguaggio *yield* al posto della notazione freccia (ma solo all'interno di un blocco di codice):

```
String season = switch(month) {
   case DECEMBER, JANUARY, FEBRUARY -> {
      String value = "winter" ;
      yield value;
   }
   case MARCH, APRIL, MAY -> "spring" ;
   case JUNE, JULY, AUGUST -> {
      String value = "summer" ;
      yield value;
   }
   case SEPTEMBER, OCTOBER, NOVEMBER -> "autumn" ;
} ;
```

Freccia vs due punti

- Possiamo ancora sostituire la notazione di freccia ->, con la notazione che si usava con lo *switch* ordinario (ovvero i due punti *:*) e usare la parola yield :

```
String season = switch(month) {
   case DECEMBER, JANUARY, FEBRUARY: yield
"winter";
   case MARCH, APRIL, MAY: yield "spring";
   case JUNE, JULY, AUGUST: yield "summer";
   case SEPTEMBER, OCTOBER, NOVEMBER: yield
      "autumn";
};
```

Freccia vs due punti

- Se usiamo i due punti invece della notazione freccia, anche con lo *switch* usato come espressione, è possibile utilizzare il fall through:

```
String season = switch(month) {
   case DECEMBER:
   case JANUARY:
   case FEBRUARY: yield "winter";
   case MARCH, APRIL, MAY: yield "spring";
   case JUNE, JULY, AUGUST: yield "summer";
   case SEPTEMBER, OCTOBER, NOVEMBER: yield
      "autumn";
};
```

Freccia vs due punti

Questo codice è valido ed equivalente agli altri esempi visti:

```
String season = switch(month) {
   case DECEMBER, JANUARY, FEBRUARY -> {yield
      "winter";}
   case MARCH, APRIL, MAY -> {yield "spring";}
   case JUNE, JULY, AUGUST -> {yield "summer";}
   case SEPTEMBER, OCTOBER, NOVEMBER -> {yield
      "autumn"};
};
```

Freccia vs due punti

Non è possibile però, mischiare le due notazioni (notazione freccia e notazione due punti) nello stesso costrutto. Per esempio:

```
String season = switch(month) {
   case DECEMBER, JANUARY, FEBRUARY: yield "winter";
   case MARCH, APRIL, MAY: yield "spring";
   case JUNE, JULY, AUGUST: yield "summer";
   case SEPTEMBER, OCTOBER, NOVEMBER ->  "autumn";
};
```

causerebbe il seguente errore di compilazione:

```
error: different case kinds used in the switch
        case SEPTEMBER, OCTOBER, NOVEMBER -> "autumn" ;
        ^
1 error
```

Poly expression (poli-espressione)

- L'espressione *switch* viene detta poly expression (poli-espressione), perché è un costrutto che può definire più espressioni.
- Ci sono due scenari da distinguere:
  - Il tipo a cui si assegna l'espressione è esplicitamente definito.
  - La parola chiave *var* è usata al posto del tipo.
- Quando il tipo che deve essere restituito dall'espressione *switch* è noto, allora tutti i case devono ritornare valori coerenti con il tipo.

Poly expression (primo caso)

```
String integer = "2";
int index = switch(integer)
{
   case "1"-> {
      byte b = 1;
      yield b;
   }
   case "2"-> {
      short s = 2;
      yield s;
   }
   case "3"-> 3;
   default -> -1;
};
```

Questo snippet sarà compilato senza errori, visto che tutti i case ritornano valori compatibili con int. Si noti che, in questo caso il compilatore si basa sulla parte sinistra della dichiarazione (LHS), per stabilire se i tipi che ritornano tutti i case sono compatibili.

Poly expression (secondo caso)

Se la variabile *index* fosse dichiarata con la parola *var*:

```
var index = switch(integer) {
{
    case "1"-> {
        byte b = 1;
        // resto del codice omesso
```

il suo tipo sarebbe dedotto come int, perché in quel caso il compilatore avrebbe controllato la parte destra (RHS) dell'espressione, anzi, delle espressioni. Siccome nelle tre espressioni sono ritornati, un *byte*, uno *short* e un *int*, ovviamente viene dedotto *int* come tipo di ritorno dell'espressione *switch*.

switch come parametro di un metodo

```
public class PolyExpression2 {
   public static void main(String args[]) {
      PolyExpression2 po2 = new PolyExpression2();
      String integer ="2";
      po2.method(
         switch(integer) {
            case "1" -> {
               byte b = 1;
               yield b;
            }
            case "2" -> {
               short s = 2;
               yield s;
            }
            case "3" -> 3;
                  default -> -1;
               }
            );
      }
      public void method(int index) {
         System.out.println(index);
      }
}
```

Exhaustiveness (esaustività)

Il compilatore non compilerà espressioni *switch* che usano un'enumerazione dove è assente una possibile clausola *case*.

```
public class Semaforo {
   public String stato;
   public void cambiaColore(Colore colore) {
      stato = switch(colore) {
         case VERDE  -> "La luce è verde";
         case GIALLO -> "La luce è gialla";
//        case ROSSO  -> "La luce è rossa";
      };
   }
}
```

Se avessimo usato lo switch come statement, il codice sarebbe stato compilabile. Quindi l'exhaustiveness è una caratteristica solo delle espressioni *switch*.

Exhaustiveness (esaustività)

Potremmo utilizzare una clausola *default* in luogo del case *ROSSO* mancante, ma tale soluzione, solo nel caso delle enumerazioni, potrebbe risultare dannosa! Potrebbe creare problemi anche nel caso volessimo aggiungere entrambe le clausole (case *ROSSO* e *default*) come di seguito:

```
public void cambiaColore(Colore colore) {
   stato = switch(colore) {
      case VERDE-> "La luce è verde";
      case GIALLO -> "La luce è gialla";
      case ROSSO -> "La luce è rossa";
      default -> "Caso imprevisto";
   };
}
```

Exhaustiveness (esaustività)

Infatti, supponiamo che l'enumerazione *Colore* si evolva per definire il colore *NERO*, che servirà per gestire le situazioni in cui il semaforo è spento:

```
public enum Colore {
   VERDE, GIALLO, ROSSO, NERO;
}
```

Quando compileremo, la clausola *default* impedirebbe al compilatore di avvertirci che non stiamo coprendo tutti i casi possibili, e il problema lo scopriremo solo al runtime.

Exhaustiveness (esaustività)

```
public class TestSemaforo {
   public static void main(String args[]) {
      Semaforo semaforo = new Semaforo();
      semaforo.cambiaColore(Colore.ROSSO);
      semaforo.stampaStato();
      semaforo.cambiaColore(Colore.GIALLO);
      semaforo.stampaStato();
      semaforo.cambiaColore(Colore.VERDE);
      semaforo.stampaStato();
      semaforo.cambiaColore(Colore2.NERO);
      semaforo.stampaStato();
   }
}
```

```
java TestSemaforo
La luce è rossa
La luce è gialla
La luce è verde
Caso imprevisto
```

Approccio alla programmazione

- Quando è il momento di iniziare un nuovo programma, l'istinto del programmatore è quello di scrivere il codice al più presto. Può sembrare giusto, ma come abbiamo già accennato nel secondo capitolo, *programmare* significa eseguire un processo come questo:
  - capire bene cosa si deve fare;
  - decidere come farlo;
  - implementare la soluzione;
  - testare il programma e con tutta probabilità tornare sul codice per risolvere eventuali problemi o apportare migliorie.

Algoritmo

- Un **algoritmo** è una successione di *istruzioni* o *passi* che definiscono le operazioni da eseguire su dei *dati* per risolve un determinato *problema*
- Molte persone identificano la programmazione stessa con il concetto di algoritmo, in realtà vedremo che la programmazione moderna non è fatta solo di algoritmi
- Il concetto di algoritmo non è esclusivo del campo informatico, e può essere applicato ad ogni campo, anche nella vita di tutti i giorni

Algoritmo

- Un esempio reale di algoritmo sono i libretti di istruzioni di montaggio di un mobile. Con una serie di istruzioni da eseguire in sequenza, essi ci permettono di realizzare un mobile usando oggetti vari (viti, assi di legno di diversa misura, vari attrezzi di lavoro etc.).
- In questo caso il problema è "costruire il mobile", i dati sono i materiali che abbiamo a disposizione, l'algoritmo è rappresentato dal libretto d'istruzioni. Una volta definito l'algoritmo, risolvere il problema significa solo eseguire le istruzioni riportate in esso. Senza libretto d'istruzioni, e senza l'adeguata esperienza, non sarà facile montare il mobile, esattamente come nella programmazione non sarà facile creare un programma senza prima decidere quali attività eseguire.

Algoritmo

- In un esempio precedente avevamo scritto:

```
if (numeroLati == 3)
   System.out.println("Questo è un triangolo");
else
   System.out.println("Questo non è un triangolo");
```

- ed avevamo anche asserito che potevamo tradurre questo codice in italiano nel seguente modo:

```
Se il numero di lati è uguale a 3
   stampa "Questo è un triangolo"
altrimenti
   stampa "Questo non è un triangolo"
```

- Si noti che se avessimo scritto un algoritmo per l'esempio in questione, questo non sarebbe stato molto differente da questa traduzione.

Algoritmo: preparare una tazza di tè

- **Problema**: preparare una tassa di tè
- **Dati di input**: tazza, acqua, bollitore elettrico, tè (varie tipologie)
- **Algoritmo**:
- riempire la tazza d'acqua sino al livello desiderato
- versare l'acqua dalla tazza nel bollitore elettrico
- accendere il bollitore elettrico
- scegliere la tipologia di tè
- posare la bustina di tè nella tazza
- quando il bollitore elettrico si spegne, versare il contenuto di acqua calda nella tazza
- aspettare i minuti necessari per la corretta infusione e rimuovere la bustina dalla tazza
- aggiungere zucchero se necessario
- **Output**: tazza di tè pronta

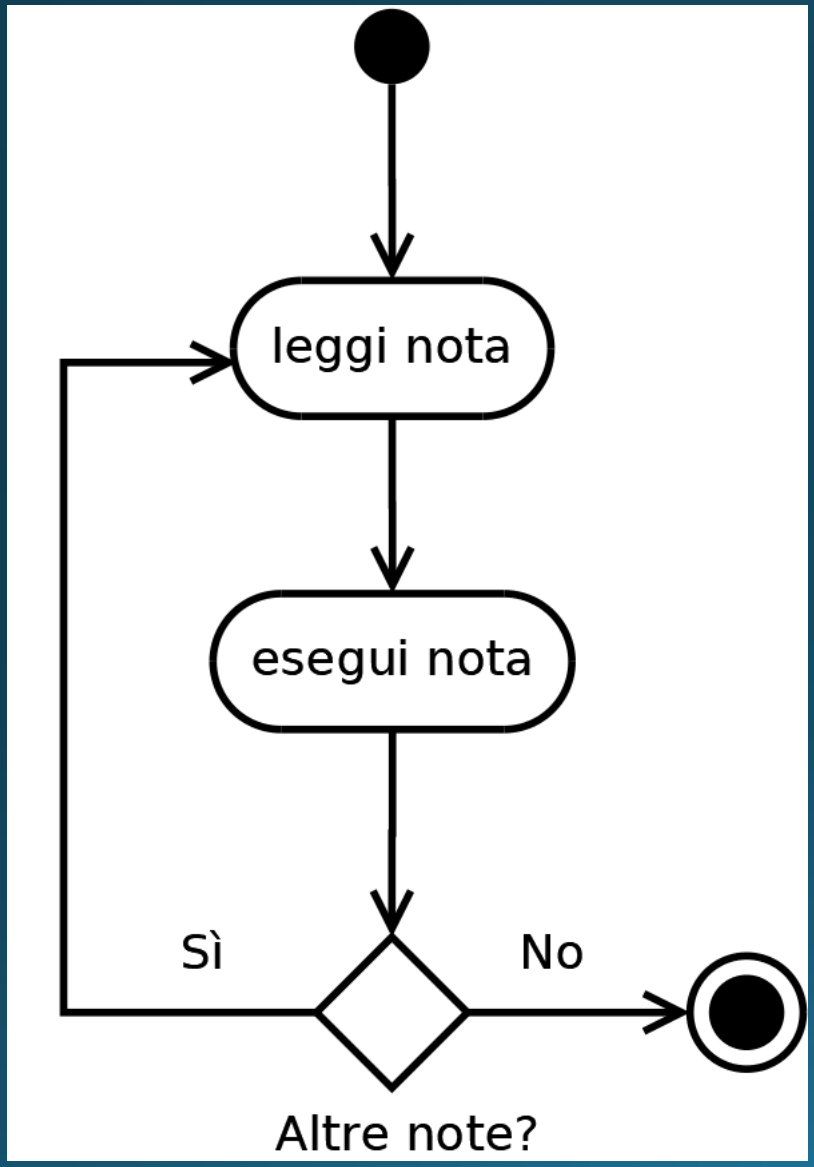
Algoritmo

- Per scrivere un algoritmo si possono usare vari strumenti, il linguaggio naturale, un linguaggio di programmazione, dello pseudocodice o anche un linguaggio di notazione.
- Scrivere un algoritmo in linguaggio naturale, sarà probabilmente la nostra prima scelta inizialmente (vedi esempio)
- Scrivere un algoritmo direttamente in un linguaggio di programmazione invece, è una scelta spesso troppo complessa, soprattutto quando si inizia
- Probabilmente utilizzare lo pseudocodice, ovvero creare l'algoritmo spiegando alcune parti con il linguaggio naturale ed altre con il linguaggio di programmazione, rappresenta il giusto compromesso
- Infine si potrebbe utilizzare un linguaggio di modellazione come l'UML per rappresentare graficamente un algoritmo

Introduzione ad UML

- Lo **Unified Modeling Language** (**UML**), è un linguaggio i cui elementi costitutivi sono per lo più grafici elementari come rettangoli, ovali, omini stilizzati, frecce e così via. Questi elementi vengono utilizzati all'interno di diagrammi seguendo certe regole.
- È un linguaggio basato sul paradigma orientato agli oggetti, e mette a disposizione diverse tipologie di diagrammi che possono essere utilizzati in situazioni e per scopi diversi.
- Più formalmente possiamo definire UML come un linguaggio che permette ad un sistema software di:
- creare specifiche: può aiutare a definire cosa deve fare il sistema;
- costruire: può aiutare a capire come deve essere fatto il sistema;
- visualizzare: consente di visualizzare il software da altri punti di vista diversi dal codice;
- documentare: con i diagrammi UML possiamo spiegare il nostro software senza leggerne il codice.

Introduzione ad UML (activity diagram)



Riscrivendo l'algoritmo in linguaggio naturale:

1. leggi la nota
2. esegui la nota
3. se ci sono altre note sullo spartito torna al punto 1

Sommario

- Conoscere e saper utilizzare i vari operatori (unità 4.1).
- Conoscere e saper utilizzare i costrutti di programmazione semplici (unità 4.2).
- Conoscere e saper utilizzare i costrutti di programmazione avanzati (unità 4.3).
- Conoscere e saper utilizzare il nuovo costrutto switch (unità 4.4).
- Comprendere che prima di iniziare a programmare è necessario prima avere chiari tutti i requisiti, e avere un piano per procedere nella codifica (unità 4.5)